NASA/CR-97- 206177

FINAL

Underwriters Laboratories Inc. ®
Programmable Systems Certification Services

RESEARCH IN THE DEVELOPMENT AND CERTIFICATION OF SAFETY-RELATED SOFTWARE

SUMMARY OF RESEARCH
17 JULY 1995 - 31 JULY 1997

Prepared under Grant Number: NAG-1-1724

for: NASA Langley Research Center

Principal Investigators: Janet Flynt and Charlotte Scheper

Submitted To:

Grants Office
NASA Langley Research Center
Hampton, VA 23681-0001

Submitted By:

Underwriters Laboratories Inc.
12 Laboratory Drive
PO Box 13995
Research Triangle Park, NC 27709-3995
Phone (919) 549-1765

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States Government nor Underwriters Laboratories Inc. nor any of their employees nor their contractors, subcontractors, or their employees makes any warrantee expressed or implied, or assumes any legal liability or responsibility for damages arising out of or in connection with the interpretation, application, or use of or ability to use any information, apparatus, product, or process disclosed, or represents that its use would not infringe on privately owned rights. This report may not be used in any way to infer or to indicate UL's endorsement of any product, or to infer or to indicate acceptability for Listing, Classification, Recognition, or Certificate Service by Underwriters Laboratories Inc. of any product or system.

Table of Contents

List of Figures

Executive Summary

Objective and Approach

The objective of this project was to evaluate leading-edge software safety research results and software certification requirements. Leading-edge software safety research sponsored by NASA Langley has focused on methods for specifying critical systems, such as the Magnetic Stereotaxis System (MSS) and the Traffic Alert and Collision Avoidance System (TCAS). In January 1994, Underwriters Laboratories Inc. published the UL 1998 Standard for Safety-Related Software and began offering an adjunct service for investigating computer-based systems that perform safety-related functions. This study investigated the results of previous NASA efforts to yield an understanding about NASA-sponsored approaches to software specification and their relationship to certification requirements for safety-related software.

As part of this project, the approaches developed in the NASA-sponsored research and the UL 1998 certification requirements were used in a case study to review a formally specified software system; i.e., a system which is specified using mathematical notation and formal logic. The Therapy Operators Console for a Clinical Neutron Therapy System at the University of Washington was selected for the case study. Part of the review of the CNTS was to look at the initial requirements document, the formal Z specification, and the code with respect to sufficiency for performing a UL 1998 investigation.

Using the results of the review of the NASA-sponsored research and the review of the CNTS, UL developed criteria for specification and design artifacts to be used in certifying safety-related software. A case study was conducted to illustrate the development and use of these artifacts. In this case study, a hypothetical design of a computer-controlled implantable device for supplying insulin was carried out. This case study and the resulting specification and design artifacts form a preliminary set of educational materials that illustrate software safety concepts and certification requirements.

In the course of this study, we reviewed three different uses of formal methods to specify safety-critical software. While each of the studies we looked at were able to demonstrate how the use of formal methods made an important contribution to their system, it is not clear from these studies how important formal methods are to certification.

Findings

Safety-related systems are generally not certified solely on the basis that they meet their specifications: they are also certified to meet the requirements of a particular safety standard. The compliance of the system requirements with respect to the requirements of the standard have to be considered. It appears that there is as yet no consensus on what should be formally specified: the safety requirements of a system, the behavior expected from the software, the design of the system, or all aspects of the system development from requirements to code. There

are also different methods for expressing and verifying the formal specification, including state diagrams which are subject to analysis using graph theory or statistical methods such as Markov analysis, and mathematical expressions and logical schema which are subject to mathematical proof.

The systems that UL 1998 primarily addresses are embedded control systems where the functionality of the code is highly dependent upon the functionality of other system components; timing and performance issues are often critical not only to required functionality but also to safety; and frequently safety issues are central to the functionality of the software (i.e., the entire control process is safety-critical). This results in two requirements for both developing and certifying such systems: (1) specification and analysis of the software system has to be conducted from end to end within the context of the complete system and (2) the dynamic behavior of the system has to be specified and evaluated for both normal and faulted conditions. This means that there are many views of the system that have to be integrated and analyzed. It does not appear that formal methods such as those reviewed in this study can address the full scope of this problem. This is due in part to the lack of methods for hierarchical development and analysis of formal specifications and the difficulty in expressing dynamic, time-dependent behaviors formally. There is also the subtle issue of the difference in specifying an intended behavior versus determining what the actual behavior is.

At whatever level of the system development process it addresses, a formal specification is only one component of the documentation required for certification. If the certification is process-based, it becomes evidence that a particular process was followed and is evaluated against the process requirements of the governing standard. If the certification is product-based, it becomes part of the descriptive information and is only one part of the evidence that a certain requirement or set of requirements for the product have been met. A formal specification is preferable to other types of specification only if it brings more clarity and precision to the system description and is more easily comprehended by a certifier, or if it provides a better linkage between all of the certification documents. Formal specifications may also have other benefits, such as fault avoidance and fault detection.

Future Research

The use of formal methods for certification is a promising area; however, the level of confidence gained by having a formal specification is affected by issues such as

- Can expected complex, dynamic behavior be described by inputs, outputs, and states to an extent that actual behavior can be determined for both normal and faulted operational conditions?

- Can safety be expressed as a property or a set of properties, given current methods and semantics, that can be proven?

- Safety and the certification of safety requires a system solution. Can an approach and supporting semantics be developed that can integrate all the different system views and support the types of analysis required by each view? Can the need to make tradeoffs between all system requirements, such as safety, reliability, performance, security, etc., be supported?

- In the U. S. safety system, certification requirements are developed through a consensus process which determines that the design and verification techniques specified in a standard are valuable and necessary. Can objective evidence be offered to demonstrate the added value of using formal methods?

Future incorporation of formal methods into the certification process will depend upon formally specifying systems in a way that facilitates the certification process.

In this report we present a view of how the many different ways of specifying various aspects of a system can be brought together into the safety certification process. This view incorporates a formal specification of each of the system components which would specify its essential characteristics (both in terms of evaluation criteria and representation styles), how it is modeled and documented, and its role in the certification process. A further formal specification would link individual models and documents across development phases with the individual requirements of the certification process. This formal specification of the components of a certification process would provide several improvements to the way in which certification is currently performed. It would provide an increased level of consistency in the application of the process to all the various systems presented for certification; it would provide clear rationale to developers for all of the documentary evidence requested by the certifier; and it would provide a consistent and well-founded certification process.

In 1996, UL established two Programmable Systems Certification Laboratories, one at the Research Triangle Park, NC office and one at the Northbrook, IL office, for research and demonstration of certification processes, tools, and techniques. The materials developed in the specification and design case study will be established in the lab and, together with the testing and analysis components already in place, will be used for demonstration and training. The lab will also provide a basis for the post-study preparation of informational seminar materials by UL. The techniques developed by NASA software safety research will be further transferred as they relate to investigations of software in safety-related products.

1 Introduction

The objective of this project was to evaluate leading-edge software safety research results and software certification requirements. Leading-edge software safety research sponsored by NASA Langley has focused on methods for specifying critical systems, such as the Magnetic Stereotaxis System (MSS) and the Traffic Alert and Collision Avoidance System (TCAS). In January 1994, Underwriters Laboratories Inc. published the UL 1998 Standard for Safety-Related Software and began offering an adjunct service for investigating computer-based systems that perform safety-related functions. This study investigated the results of previous NASA efforts to yield an understanding about NASA-sponsored approaches to software specification and their relationship to certification requirements for safety-related software, such as

- Does the NASA-sponsored research provide evaluation criteria to help us assess the consistency between the links in the chain from requirements to code?

- How does a formal specification relate to the code as implemented?

- What is available from the NASA-sponsored research regarding properties of formal specification that should be considered during certification?

The approaches developed in the NASA-sponsored research and the UL 1998 certification requirements were used in a case study to review a formally specified software system. The Therapy Operators Console for a Clinical Neutron Therapy System was selected for the case study. Part of the review of the CNTS was to look at the initial requirements document, the formal Z specification, and the code with respect to sufficiency for performing a UL 1998 investigation.

Based on the results of the research review and the case study of the CNTS, a case study in the design of a computer-controlled implantable device for supplying insulin was created and used to develop specification and design artifacts that could be used for a preliminary set of educational materials illustrating software safety concepts and certification requirements. It is intended that the preliminary materials will provide a basis for the post-study preparation of educational engineering seminar materials by UL. The seminar materials will further communicate NASA software safety research results as they relate to investigations of software in safety-related products.

2 Review of Software Safety Research

The review of NASA-sponsored software safety research on the TCAS and MSS systems concentrated on the use of formal methods. Sections 2.1 and 2.2 describe the formal specification concepts used in TCAS and MSS, respectively.

Integrating the results of the review of the work by Leveson on TCAS with that of Knight on MSS, we find

- state machine models, graphically represented
- a structure for analyzing requirements for real-time, process-control software
- an emphasis on specifying behavior
- formal verification of properties of models
- an emphasis on a separate safety specification
- incorporation of fault handling and failure response in models
- use of analysis of possible risks to derive requirements
- demonstration of communication and verification roles of specification
- formal derivation of evaluation criteria and test sets from specifications.

The underlying theme in both cases is the use of models to describe intended behavior. The models incorporate a formal semantic that stipulates the required properties of the model and how a particular model can be analyzed and proven to satisfy those properties. A review of formal methods literature for high assurance systems indicated that the methods used in these cases were somewhat less formal than those that use mathematical notation and are based on logical systems such as predicate calculus. However, the methods used in TCAS and MSS are clearly more easily understood by non-specialists in formal methods. This increased clarity would seem to improve their value during system development and during certification.

2.1 Formal Specification Concepts from TCAS

The Traffic Alert and Collision Avoidance System (TCAS) II is an aircraft collision avoidance system. A Minimal Operational Performance Standards (MOPS) document was produced and adopted in 1983, but was found to be deficient and to not provide the real system and software requirements needed for FAA certification [Leveson94]. Nancy Leveson and a group at the University of California at Irvine specified the system requirements from the existing pseudocode representation of TCAS II logic [Britt94], and this specification was adopted as the official TCAS requirements specification [Leveson94].

The basis of the Leveson team's approach to specification is an abstract state-machine model and a set of criteria by which analysis procedures can be defined to provide semantic analysis of real-time, process-control software requirements [Jaffe91]. These criteria identify missing, incorrect,

and ambiguous requirements, ensuring (1) completeness of transitions and default values, (2) complete specification of inputs and outputs, (3) complete specification of computer-operator interactions, (4) complete description and handling of inputs, (5) complete specification of output conditions with respect to timing and value, (6) complete specification of the relationship between inputs and outputs, and (7) complete specification of the paths between states with respect to certain properties [Modugno97]. The important aspects of this approach for this study are that

- it models behavior, including assumptions about the behavior of other system components
- it provides formal analysis as well as formal description
- it includes in the model information that is usually captured in several different models
- it focuses on process control, including process state, controlled variables, corrective actions (outputs), and prediction of future states.

The formal state machine model is called Requirements State Machine (RSM). The specification language is Requirements State Machine Language (RSML) and is an outgrowth of Statecharts. The RSML allows the specification to be described in a way that is more easily understood than mathematical notations while the underlying RSM model provides the formalisms required for formal analysis [Leveson94].

A case study documented in [Britt94] concluded that the Leveson team had demonstrated the feasibility of specifying a large safety-critical system using reverse engineering and a formal specification language. This case study also examined the TCAS work in terms of the role of formal methods in the certification of safety-critical systems. According to [Britt94], the role of formal methods was determined to be to drive all of the qualitative assurance components of safety certification. In this view, random testing drives quantitative assessment, and together qualitative and quantitative assurance comprise certification. The basic characteristic of formal methods is a formal specification that precisely describes the system in a manner amenable to mathematical analysis. The TCAS project found that one of the problems in using formal specification languages is that it is difficult for non-mathematicians to understand the resulting specifications. Thus, while a formal specification may be for formal analysis, it has certain disadvantages for validation. RSML addresses this problem by providing an interface that can be understood easily. RSML is translated to RSM, the underlying math model, for formal analysis.

[Britt94] concluded that the formal methods approach used in TCAS II was feasible, but that its effectiveness was limited by a lack of automated tools and a lack of guidelines for writing and using RSML specifications. According to [Leveson94], there are still some problems with this approach in the areas of overall event sequencing and synchronization. Some problems with proliferation of states were also noted. An interesting observation made by Leveson is that there

is a need for multiple types of notation, including graphical, symbolic, tabular, and textual, and that it is likely that the type of information to be conveyed will determine the type of notation.

Subsequent to the development of the TCAS II specification, methods were developed to automatically analyze formal, state-based requirements specifications for d-completeness (a response is specified for every possible input), consistency (no conflicting requirements), and determinism [Heimdahl96]. The analysis algorithms and tools were then validated on TCAS II. The results of applying the analysis techniques were compared to the results of an extensive independent verification and validation of the RSML TCAS specification and it was found that inconsistencies found during verification and validation were found by the analysis and that some subtle inconsistencies were found by the analysis but not by the verification and validation [Heimdahl96].

2.2 Formal Specification Concepts from MSS Case Study

The Magnetic Stereotaxis System (MSS) [Knight93B] is a device for performing neurosurgery via an externally applied magnetic field. The computer system controls several physical subsystems, combines and displays X-ray images with Magnetic Resonance Images based on inputs from the neurosurgeon. The MSS is a safety-critical system and the failure of any of the physical subsystems or of the control software could result in patient injury. In a NASA-sponsored project, the MSS is being used by Dr. John Knight at the University of Virginia as a case study to evaluate a comprehensive approach to software safety that he has been developing.

Dr. Knight's research [Knight93A] focuses on the specification of safety and the subsequent verification of the correct implementation of the specifications. A systems engineer prepares a set of specifications for the software and the software is defined to be safe if it complies with these specifications. Thus, in Knight's framework, demonstration of software safety is an exercise in verification and software is safe to the extent that verification is successful.

In [Knight93B] a software specification is said to consist of (1) the intrinsic-functionality specifications (what the software is to do during normal operations), (2) the failure-interface specifications (what interface the software will present to other system components in the event that it has failed and can not provide its functionality), and (3) the recovery-functionality specifications (the functionality required of the software when another system component has failed). The failure-interface and the recovery-functionality specifications are said to comprise the software-safety specifications. Software is defined to be safe if it complies with its safety specification, given that the specification is complete and accurate and the implementation implements the safety specifications correctly.

The next area addressed in [Knight93B] is the development of the specification. Knight asserts that the specification must be developed according to a "rigorous, repeatable process". His

approach to this process is based on system fault trees. In this approach, software functions are first added to mitigate risks of failures and then examined and constrained to limit their risks. The results of these two successive phases are the recovery-functionality and the failure-interface specifications.

According to [Knight93A], the improvement that this approach makes over other approaches is that it stresses the importance of a separate specification of software safety requirements. Several advantages are claimed for having a separate specification, including that it

- makes verification possible
- provides a basis for precise communication between the software engineer and the systems engineer, and
- permits exploitation of the precise definition of software safety in his definitional framework.

Knight emphasizes that verification is a key aspect and has previously demonstrated the concept of deriving a set of test cases from formal safety specifications. If a test set could be derived and shown to be complete, the correct execution of the test set would constitute proof that the safety specification was met by the software implementation. He states that the argument against establishing correctness of safety-critical software by testing is a "general result to which there are exceptions" and that safety is one of these exceptions because (1) safety specifications are typically smaller than functional specifications and (2) time can be accelerated during testing of safety facilities since they are called upon to act infrequently during normal operation.

Dr. Knight is developing a process to develop software specifications in a rigorous and complete manner, including developing support techniques for the process. He identifies three key support techniques: information flow analysis, software failure emulation, and tri-state models. The combined output of information flow analysis and software failure emulation is the complete software safety specification. Descriptions of these techniques can be found in [Knight93A].

Knight's work on MSS has pursued developing a rigorous approach to determining and specifying software safety requirements and on means of assessing software safety using both formal verification techniques and empirical testing methods. A formal specification of software safety requirements is the enabling connection between these two components of the work. Knight has specified ways of both determining the safety requirements and evaluating that the requirements are met.

3 Case Study: Review of a Formally Specified Software System

The Clinical Neutron Therapy System (CNTS) is a radiation therapy system which has been in operation at the University of Washington since 1984 and for which a new control system is being developed [Jacky95c]. Its control system includes six processors, processes over one thousand input and output signals, and has to meet demanding requirements for availability, equipment protection, and human safety. [Jacky90]

For the new control system, a team led by Jonathan Jacky and Ruedi Risler developed an "informal" specification using prose, tables, diagrams, and formulas. This specification is documented in Jacky90, Jacky92, and Jacky95a. A team led by Jacky also developed a formal description in Z of one of the subsystems of CNTS, the therapy console control program, which is used to set up and deliver the patient treatments. This formal description serves as the detailed design. It was validated against the text by inspection and the code was derived from it, some formally and some informally [Jacky94a, Jacky95b, Jacky95c, Jacky96,]. The Z specification is documented in Jacky95d.

As part of this study, UL addressed the applicability of UL 1998 to medical devices such as CNTS. This review consisted of two parts: a desk review of the CNTS documentation and an on-site review of the system. It focused on the therapy console control program. This part of the system is described in Section 3.1; the use of formal methods in the development is described in Section 3.2; our conclusions regarding requirements for submission of evidence for certification are documented in Section 4.2, after a discussion of our conclusions regarding certification requirements.

3.1 CNTS System Description

According to [Jacky90], the entire control system is essentially divided into a proton beam control system and two treatment control systems. The difference between the two treatment control systems is that one has a rotating gantry whereby the beam can be moved around the patient and the beam shaped by a leaf collimator, whereas the other has a fixed beam. Several loosely-coupled subsystems comprise the control system; each subsystem has its own inputs, outputs, processors and memory. We are concerned in this study with the therapy control system and the user interface.

The therapy console control program provides the user interface to the CNTS control system, controls the physical components of CNTS that deliver the treatment, and helps to ensure that treatment is delivered in accordance with the particular prescription for a patient [Jacky96]. Each prescription is stored in a database and consists of several different beam configurations, or fields. A field is defined by about 50 machine settings, which must be set correctly for the prescribed treatment to be delivered. Some of the settings are automatic and some are manual.

The therapist uses the console program to choose fields from the prescription. The program checks all settings against the prescription and ensures that the beam cannot be turned on until the correct settings have been achieved. The beam is turned on by a separate, nonprogrammable mechanism after the program indicates that the system is ready.

The specification of the user interface is documented in Jacky92. The functional requirements for the user interface are

- a timely response whenever the user interacts with the system
- the user can determine that the system detected the user's action(s)
- something on the display should always update immediately, even if operation takes "appreciable" time
- system responds promptly and appropriately to invalid or erroneous input (does not ignore it).

The only requirements for screen design are

- required data items to be shown
- consistency among screens with respect to style.

Each screen consists of 6 regions; there are 3 kinds of display and 3 common operations.

The system is of course safety-critical. There are risks of radiation exposure, collisions between people and physical structures of the CNTS, and falls into a pit in the floor which is exposed when the gantry rotates. The primary radiation exposure risks are from irradiating the wrong volume (missing the target) and delivering the wrong dose. These risks are mitigated in several ways, including the use of nonprogrammable mechanisms to establish generic safety conditions, visual checks by the therapist, the establishment and verification of safety conditions by the control program, and elements of the physical design of the system that limit the radiation dose to "medically reasonable values" [Jacky96].

The requirements for detecting and responding to faults and errors are described below:

User Input Errors

- prompt, appropriate response (no ignore) to invalid or erroneous input
- range checking on numerical inputs and rejection of out-of-range inputs

Failed Operations and Equipment Faults

- equipment fault that prevents successful completion of required operation results in message in log region and in log file which is time stamped and describes problem
- terminal displays show faults and interlocked conditions
- safety mechanism provided by interlocks
- fault in equipment does not stop control system

Control System Errors

- can detect certain failures such as communication failures between main control computer, terminals, and other computers in the control system
- special system error display in operations window when failure is detected
- operators should not attempt to proceed
- manual of recommendations
- some errors do not preclude continued operation
- interlocks to disable motions and prevent beam from being turned on are attempted to be set.

The Therapy Operations Terminal controls 5 subsystems: Gantry/PSA, Filters/Wedges, Leaf Collimator, Dosimetry, and Therapy Interlocks. It performs fourteen operations: write log message, logout, select patient, select field, store field, experiment mode, field summary, gantry/PSA, filters/wedges, leaf collimator, dosimetry/therapy interlocks, edit, override, auto setup. A typical treatment sequence consists of the following steps:

- operator invokes select patient
- operator invokes select field (if prescribed settings don't match actual settings, all five therapy subsystems are *NOT READY*
- operator invokes field summary
- operator invokes auto setup (leaves, filters and dosimetry system automatically set up)
- operator enters treatment room and manually sets up external motions
- operator leaves treatment room and closes the door (if setup is accurate and cyclotron running well, all subsystems are *READY*)
- operator presses start button; run begins
- after dose has been delivered, dosimetry system turns beam off and dosimetry/therapy interlocks subsystem becomes *NOT READY*

Specifications for the control of the gantry, filter/wedge, leaf collimator, dosimetry, and proton beam controllers by the therapy console control program are documented in [Jacky95a]. Of primary interest are the mechanisms for detecting and responding to errors. Each controller has a protocol and the control program can determine if a controller's behavior conforms to its protocol. If there has been an error, and a controller does not perform according to its protocol, it is required that the control program not halt and not hang. Other functions that are independent of the "failed" controller can continue to work normally. In the event of an error, a software interlock is also set and the corresponding control subsystem becomes *NOT READY*. The therapy interlock relay in the HSIS (Hard-wired Safety Interlock System), which is a chain of switches that have to all be closed for the beam to be turned on, opens; enable relays are de-energized to disable any motions driven by the controller; and a message is displayed and logged.

If the controller follows its protocol and issues no error messages, but the controlled settings do

not reach the commanded values, the problem is assumed to be an equipment error, not a controller error. In this case, the controller interlock is not set, but the settings will not be read and the check and confirm software interlock and the therapy software interlock will be set, disabling the beam from turning on.

The interlocks are the primary safety mechanism, ensuring that, in the event of an error, the beam and equipment motions are disabled. There are both hardware and software interlocks. The hardware interlocks are physical devices such as microswitches or relay contactors; the software interlocks are binary variables with the values set and clear.

Concurrency issues arise since the system controls multiple devices. Input from one device can be received concurrently with the issuance of command controls to another and some devices can signal unsolicited events requiring prompt response. However, there are no hard real-time requirements as the typical response requirement is in the range of tenths of a second [Jacky96].

The development process for the control program consisted of the following steps [Jacky96]:

- an informal specification was produced using prose, tables, diagrams, and formulas
- a formal description in Z was produced
- the Z description was validated against the informal specification by inspection
- code was derived from the Z description; most code was derived informally, but some was formally derived
- code was verified by inspection (reviews) and testing.

Testing consisted of functional, stress, and acceptance tests.

3.2 Use of Formal Methods in the Development of CNTS

The formal description for portions of the control program was written in Z. This description is a model that expresses the behavior and serves as a detailed design [Jacky96]. It models most of the program internals, including field settings and safety interlocks. It does not model program features like the appearance of the user interface displays. According to Jacky94a, complex behaviors of the whole system "emerge" from the composition of simpler operations. Differing degrees of formal rigor were used in developing different portions of the program; some code was translated from Z to Pascal by inspection, but the transition procedure was formally derived [Jacky95b].

The formal Z texts are an intermediate description between the informal, prose specification and the actual code. It is used to guide coding and to perform reviews. In fact, the code was derived directly from the Z texts: Z paragraphs correspond to data structures, functions, and procedures in the code [Jacky96].

Jacky states that the formal text corresponds to two dimensions of the development process:

partitioning and refinement. It partitions the system into independent subsystems or modules and describes simple operations on each; however, the delegation of functions among software and hardware components had been completed before the Z description was begun [Jacky94a]. It refines the system from an abstract model to a detailed design.

The safety requirement for the control program is "the beam can only turn on when the actual setup of the machine is physically safe, and matches a prescription that the operator has selected and approved" [Jacky94a]. To meet this requirement, the control program has to establish and confirm the *SafeTreatment* condition; thus, "clearing the master therapy interlock is the central safety-critical act of the program" [Jacky94a]. Additionally, the control program logs every attempt to treat a patient and indicates certain faults in the machinery or environment, such as unreasonable sensor readings and inaccessible files [Jacky94b].

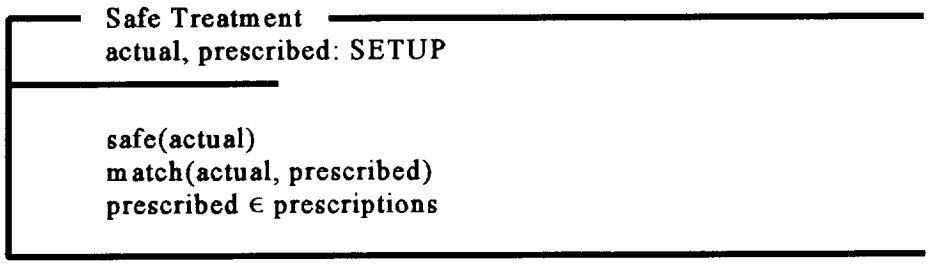

Figure 3.1 Formal Statement of Safety Requirement [Jacky94b]

The safety requirement is expressed formally in Z as shown in Figure 3.1 [Jacky94b].

The *SafeTreatment* condition is checked periodically; if it is found to be true, the interlock is cleared and the beam can be turned on. If the *SafeTreatment* condition becomes false at any time, the interlock is set and the beam is turned off.

The control program was partitioned into the subsystems for console, session, field, interlock, display, timer, files, and controllers, and operations were defined on each subsystem. Thus, the operations on the whole system are defined as separate operations on each affected subsystem, and system behavior "emerges" by composing operations [Jacky94b].

There are only 2 explicit real-time constraints, which are specified in Z as ordinary state variables [Jacky94b]. These are timeouts and expirations. A timeout constraint states that a setting must be achieved by the controller within a deadline that is measured in seconds. An expiration constraint states that interlocks must be set if certain conditions and calibrations expire and is measured in from minutes to hour. Also, small delays are tolerable in signaling timeouts and expirations.

Jacky states in [Jacky95c] that this Z specification considers some "discontinuous features" not usually considered by classical control theory, such as safety interlocking. To accomplish this he has proposed a method for calculating interlock conditions for particular operations from system safety assertions. Further, he states that this application is an open-loop system, which is not as well discussed in control theory literature as closed, feedback loop systems.

In [Jacky95c], he presents the CNTS Z specification as a partially complete framework for specifying these types of systems. The basic assertion of his framework is that a system is a collection of state variables that must obey certain control laws and safety assertions. The system is modeled by a Z state schema, where state variables are named in schema declarations, control laws and safety assertions are schema predicates, and safety assertions are formulae that place additional constraints on state variables. Operations change the values of some state variables and are modeled by Z operation schema. Interlocks to prevent operations proceeding if a risk of potential hazard exists are modeled as preconditions for operation schema. To implement this model of interlocks, he states in [Jacky95c] that he developed a stronger version of the usual Z precondition by creating a schema that consists only of declarations of state variables whose values cannot be directly controlled.

In the course of developing the Z models, Jacky noted that this framework was not practical because all system state variables appear in a single state schema and real process control systems have a large number of state variables and result in a large number of operations [Jacky95c]. To get around this limitation, Jacky recommends identifying when similar components are repeated and treating each kind of component as an abstract data type or object [Jacky95c].

For CNTS, they have found several types of components to be useful, including analog control parameters, power supplies and servometers, and discrete indicators (i.e., system level state variables that don't belong to any obvious component). The full system is expressed by combining component. Since the state variables and predicates are contained within the components, the state schema at the system level can be much smaller [Jacky95c].

Jacky also found [Jacky95c] that he had to use Z idioms, or constructions that are outside normal Z usage. These idioms include promotion, which makes operations at component-level available at system level, and operations on multiple components, which are system level operations accomplished by performing the same method on multiple components.

Jacky did not at first model a "main" program that controlled the execution of the various operations: each top-level operation schema executes when its preconditions are satisfied and its input becomes available. He later added some constructs to model execution control. These constructs are described in [Jacky 95b] and are summarized below.

The operations are modeled as one or more Z operation schema on the *Console* state and have state variables; they are either available or engaged. Input events are modeled by *Event* operation schema: input events cause changes in operation state. The overall control structure is modeled as $ConsoleOP =^ SelectDisplay \vee EditMessage \vee ... \vee IgnoreOthers$. *ConsoleOP* defines a finite-state machine where each of the constituent operations describes a single transition.

Jacky also derived a state transition table from Z texts and developed code to interpret state machine in this format. The table is built by listing all operation schema that are combined in *ConsoleOP* and extracting the preconditions of each. The table is described formally as follows: each entry is described by a *Transition* schema; indentations in rows are indicated by nesting *depth; state* and *input* preconditions by predicates expressed as unary relations on *Console* and *INPUT*, respectively; *operations* are modeled by functions on *Console* states. The *transition* function takes any *Console* state and input into a new *Console* state; traverses the table from top to bottom, looking for an enabled entry. When an enabled entry is found, its input precondition is tested: if the precondition is satisfied, the transition has triggered; if no entries are triggered, the new state is the same as the old state.

Each operation schema is implemented directly from the state transition table as 3 separate units: a boolean function to test preconditions on state variables, a boolean function to test input preconditions, and a procedure to implement the change of state. The interpreter reads the user's input and invokes the functions and procedures as directed by the state transition table. The state transition table is implemented as an array of records. The transition function is a procedure without parameters and tests each table entry in turn. The code is correct if program variables satisfy their formal definitions when the assignment is made to the unary prefix relation on table entries that specifies if a table entry is enabled. The dispatcher is a loop that removes one event from the head of the event queue and calls the transition procedure.

In [Jacky95b], Jacky states that he found his use of Z to offer advantages for partitioning the design and factoring out recurring features. As a result, they were able to split out the generic part of the implementation, which is the event handling and dispatching, from the application-specific part, which is the state transition table and the functions and procedures that implement the operation schema. He believes that this will allow them to develop control code for the other parts of the system by replacing only the application-specific portion of the code.

Compared to Knight and Leveson, Jacky has developed a formal specification at a lower level in the chain from requirements to code. His primary purpose has been to derive the development of code in such a way as to guarantee that the safety requirement that the computer-controlled settings match the prescription for the patient is met. His specification is more formal than the

other two in that it uses a more mathematical notation. However, he used no formal verification techniques. Like the others, Jacky's specification is state-based. The triggering mechanism is not integral to the semantics of the Z model as it is in the other two cases and had to be described explicitly as an additional element of the model.

4 Certification Requirements

In the course of this study, we have reviewed the use of formal methods to specify safety-critical software. While each of the studies we looked at were able to demonstrate how the use of formal methods made an important contribution to their systems, it is not clear from these studies how important formal methods are to certification.

Safety-related systems are generally not certified solely on the basis that they meet their specifications: they are also certified to meet the requirements of a particular safety standard. In other words, the compliance of the system requirements with respect to the requirements of the standard have to be considered. It appears that there is as yet no consensus on what should be formally specified: the safety requirements of a system, the behavior expected from the software, the design of the system, or all aspects of the system development from requirements to code. There are also different methods for expressing and verifying the formal specification, including state diagrams which are subject to analysis using graph theory or statistical methods such as Markov analysis, and mathematical expressions and logical schema which are subject to mathematical proof.

The systems that UL 1998 primarily addresses are embedded control systems where the functionality of the code is highly dependent upon the functionality of other system components; timing and performance issues are often critical not only to required functionality but also to safety; and frequently safety issues are central to the functionality of the software (i.e., the entire control process is safety-critical). This results in two requirements for both developing and certifying such systems: (1) specification and analysis of the software system has to be conducted from end to end within the context of the complete system and (2) the dynamic behavior of the system has to be specified and evaluated for both normal and faulted conditions. This means that there are many views of the system that have to be integrated and analyzed. It does not appear that formal methods such as those reviewed in this study can address the full scope of this problem. This is due in part to the lack of methods for hierarchical development and analysis of formal specifications and the difficulty in expressing dynamic, time-dependent behaviors formally. There is also the subtle issue of the difference in specifying an intended behavior versus determining what the actual behavior is.

At whatever level of the system development process it addresses, a formal specification is only one component of the documentation required for certification. If the certification is process-based, it becomes evidence that a particular process was followed and is evaluated against the process requirements of the governing standard. If the certification is product-based, it becomes part of the descriptive information and is only one part of the evidence that a certain requirement or set of requirements for the product have been met. A formal specification is preferable to other types of specification only if it brings more clarity and precision to the system description and is more easily comprehended by a certifier, or if it provides a better linkage between all of

the certification documents. Formal specifications may also have other benefits, such as fault avoidance and fault detection.

4.1 UL 1998 Certification Requirements

The UL 1998 Standard [UL1998] specifies requirements for safety-related software. Considered as a whole, UL 1998 can be said to specify general requirements for a safety policy that are made specific to the software under investigation by the safety requirements of the end product standard and the risk analysis conducted by the developer. These requirements address and provide constraints for a broad spectrum of software and system considerations in addition to the usual design considerations, including the following:

- the impact of changes to software
- the management of any designed variability in the operational configuration of the software
- the integrity of critical sections of code
- logical and physical partitioning of software
- execution of the code
- results of failures
- the prevention, detection, and resolution of faults
- measures to address hardware failures
- version control and identification
- initialization and termination of software and the controlled device(s)
- operational considerations such as memory conflicts, priority and scheduling conflicts, and procedures for initiating sequences related to risks
- system issues
- user interface issues.

Currently, to show compliance with the UL 1998 requirements, particular kinds of documentation must be presented. The standard itself calls for documentation of software and system state diagrams, a FMEA, a Fault Tree, design descriptions, and descriptions of fault modes and fault prevention, detection, and resolution methods. From our review of the CNTS and our case study in the design of an artificial pancreas, we have identified the following set of requirements and specification that need to be provided:

- hazard/risk analysis
- safety requirements
- functional requirements
- system requirements
- software safety requirements
- end-product safety requirements

- specification of the system environment
- specification of the model of expected faults and failures
- specification of the requirements for fault prevention, detection, and resolution.

We have also identified the following set of system descriptions that need to be provided:

- system description
- software description
- operating system description
- microprocessor description
- structural diagram of the system
- structural diagram of the processor
- architectural diagram of the software
- map of software architecture to the structural diagram
- computational model
- functional model
- behavioral model
- FMEA

Further, all of the various pieces of documentation must be complete and consistent with respect to describing the safety function and risk profile of the software being investigated. The goal is to be able to follow a thread through the documentation from risk analysis to implemented code for each risk identified. Thus, software states must be mapped to system states, software components mapped to microprocessor components and to the components of the end product that it controls, monitors, or otherwise interacts with, and the identified risks must be mapped to each of these.

4.2 Examples of Documentation for Certification

Developers of a system are developing their documentation as work on the system proceeds. They rely on it both to document the decisions that they have made and to guide the design and implementation process. Whatever form the documentation takes, the developers have an intimate familiarity with it both in terms of the details of the documentation and how to "index" into the various documents to locate information on particular aspects of the system. The certifier is in an entirely different position. He has no familiarity with the system, but has to come to an understanding of the system functionality, its architecture, its safety requirements, its mechanisms for fulfilling its safety requirements, and the methods and processes used in its development. Therefore, there is a need for documentation directed toward certification, which could be referred to as certification documentation.

The general characteristic of this documentation is that it organizes complexity in an easy to comprehend style. To illustrate some of the concepts, we discuss three types of certification documentation in this section that provide necessary system and safety context for safety-related control software. We illustrate each type of documentation with examples we created for CNTS.

Mapping of Hazards to System Components

UL 1998 requires that a risk analysis be conducted to identify all possible risks, that system components associated with those risks be identified, and system and software states be identified and analyzed with respect to risk. This provides the certifier with the safety context for the systems, allows the certifier to determine where the focus of the evaluation should be, and provides the starting point for tracing safety concerns through the system.

Having identified in a risk analysis what the hazards of the system are, the aspect of the system that constitutes the hazard needs to be traced through the system at a level detailed enough to identify the components associated with that aspect of the system. In the case of CNTS, the beam is the source of the radiation hazard. Figure 4.1, the Beam Flow Diagram, traces the beam through the physical components of the delivery system that affect the beam level, its distribution and intensity, and the target area. These physical components affect the dose administered and the area treated. The diagram also includes the moving surfaces external to the beam apparatus that affect the selection of the target treatment area.

Risk analysis is a common activity in developing safety-related system; but mapping the risks to system software components is not. Figure 4.2 contains a tree structure which identifies the potential hazards and associates them with the physical components of the treatment delivery system and their related control systems. This is an abstract look at the information contained in a mapping of hazards to system components. Note that the physical components in the risk tree correspond to those in the beam flow diagram. We would like to be able to associate a hierarchical breakdown of software modules and data variables to each component of the Beam Flow Diagram so that the hazards tree can be extended to include individual software modules and data structures.

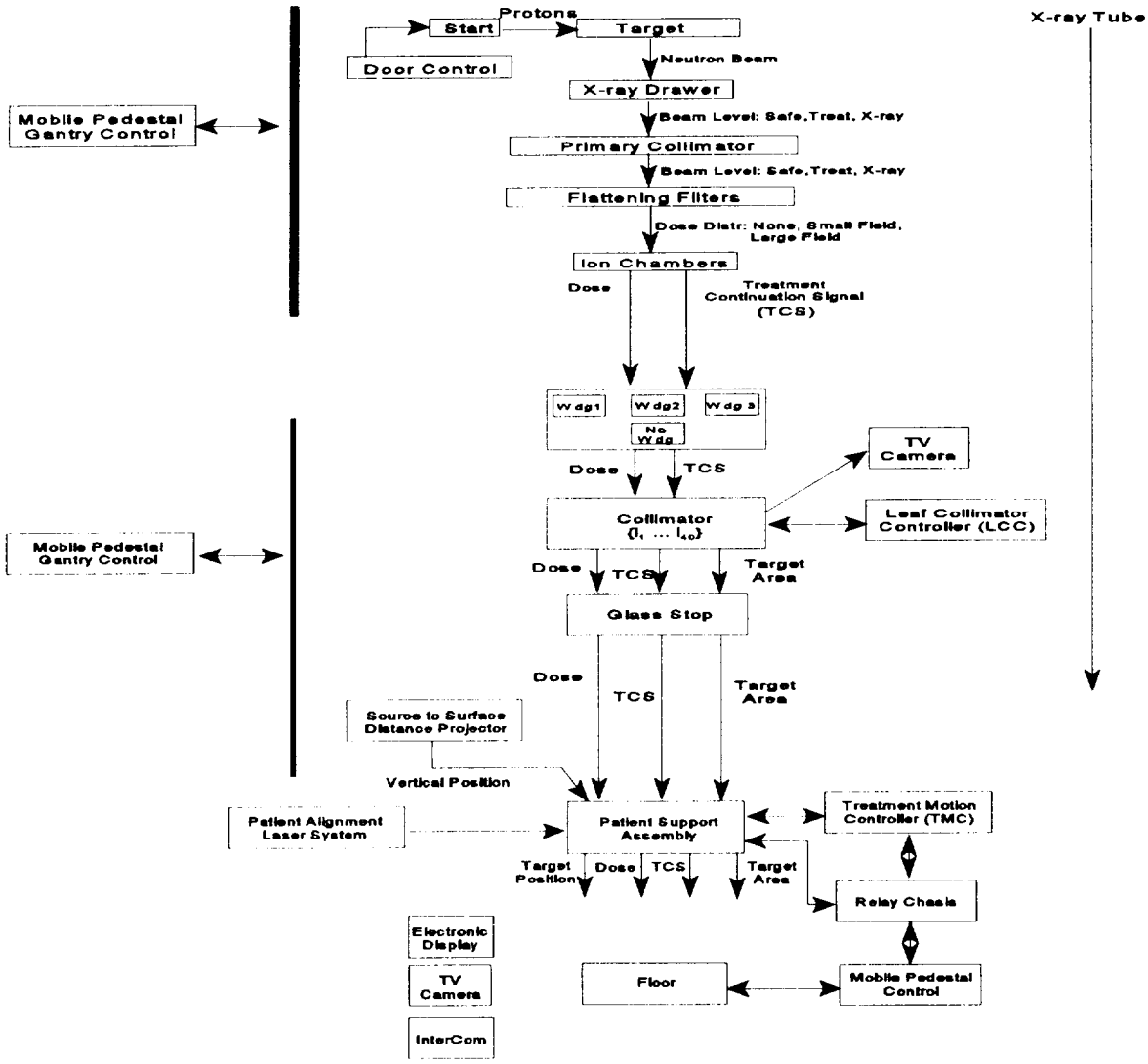

Figure 4.1. Beam Flow Diagram

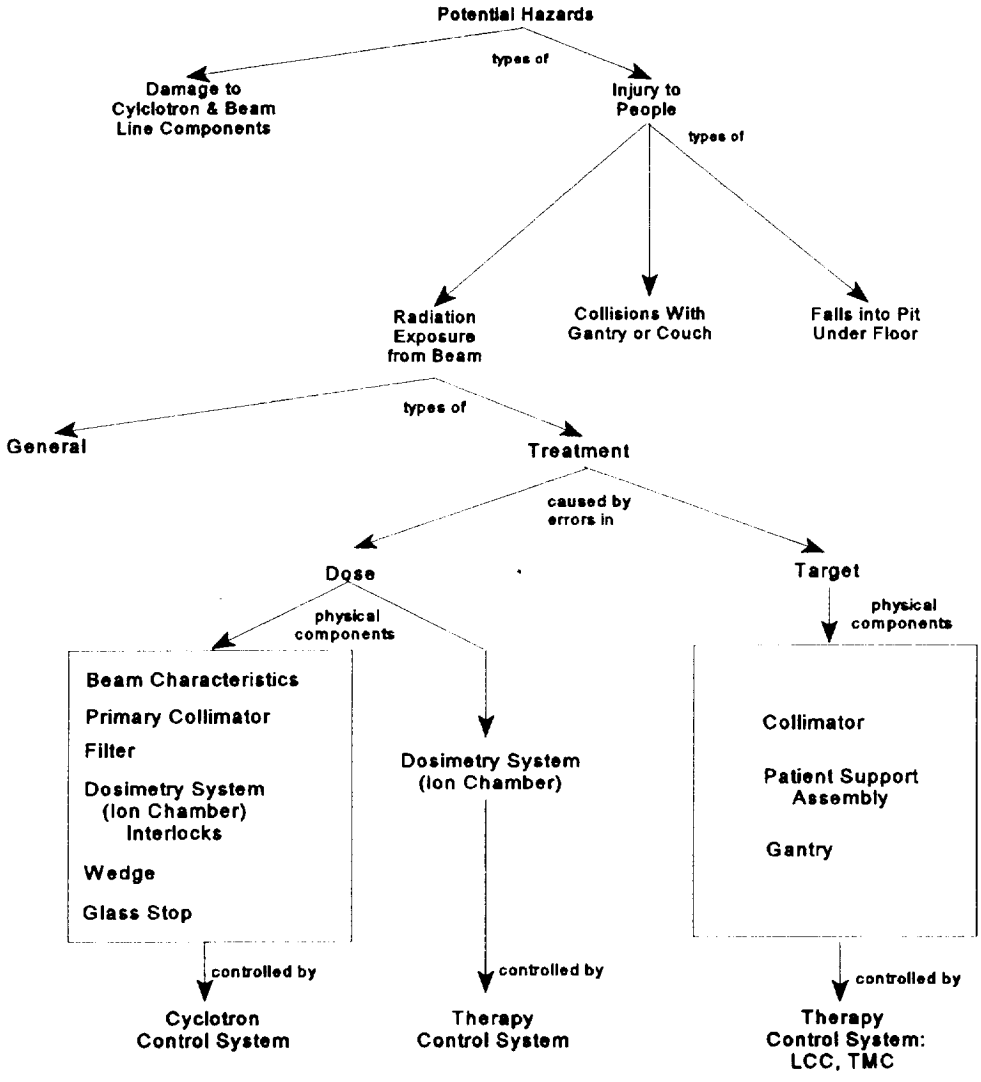

Figure 4.2. Hazard to Component Mapping

System Architecture

Having documented the hazards associated with the system and traced its hazardous aspects through the system, we now need to identify the software systems and show how they interact with the physical components of the system and the users of the system. To do this, we have created a hierarchical architecture model, shown in Figures 4.3 through 4.5. This view of the architecture of the system indicates the system components and the connectivity and relationships between them. At the top level (Figure 4.3), the major data groups and the users are identified, and the control loop between the control system and the physical system is indicated. The next level (Figure 4.4) expands the view of the control system to show its major components, who the users are, the mechanisms they use to interact with the control system components, the aspects of the physical system controlled by the control system components, and the computer systems on which the control system components are executed. At the next level (Figure 4.5), the treatment control system is expanded to show the subsystems which comprise it. Each of these subsystems can in turn be expanded to indicate its functions, and the related physical and software components.

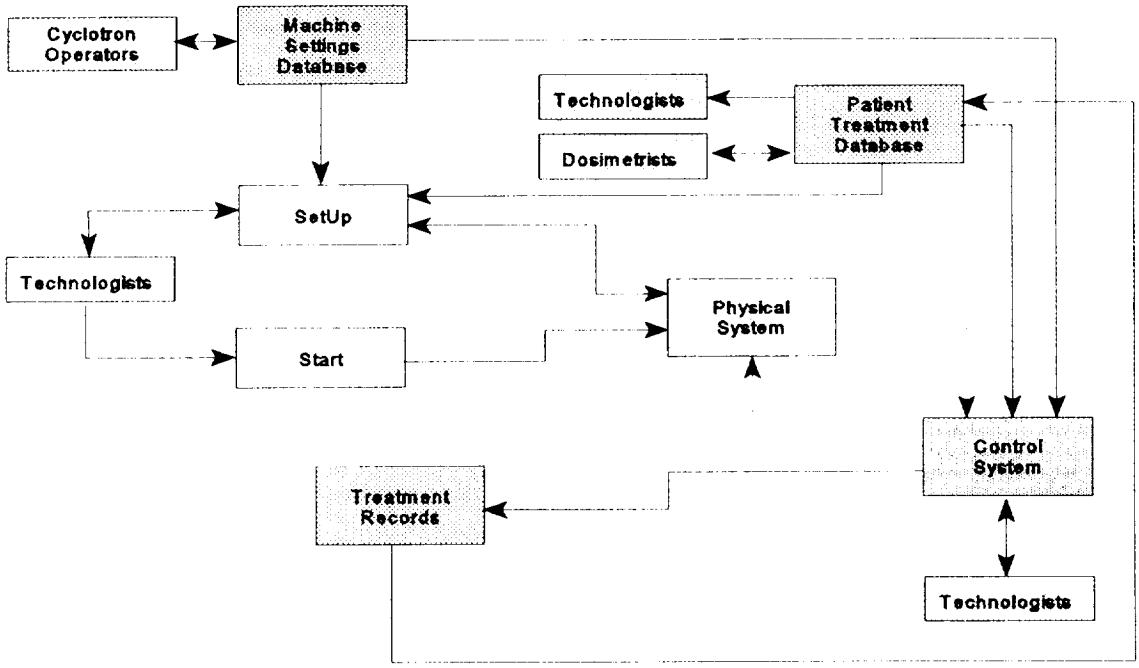

Figure 4.3. Top-Level Architecture View

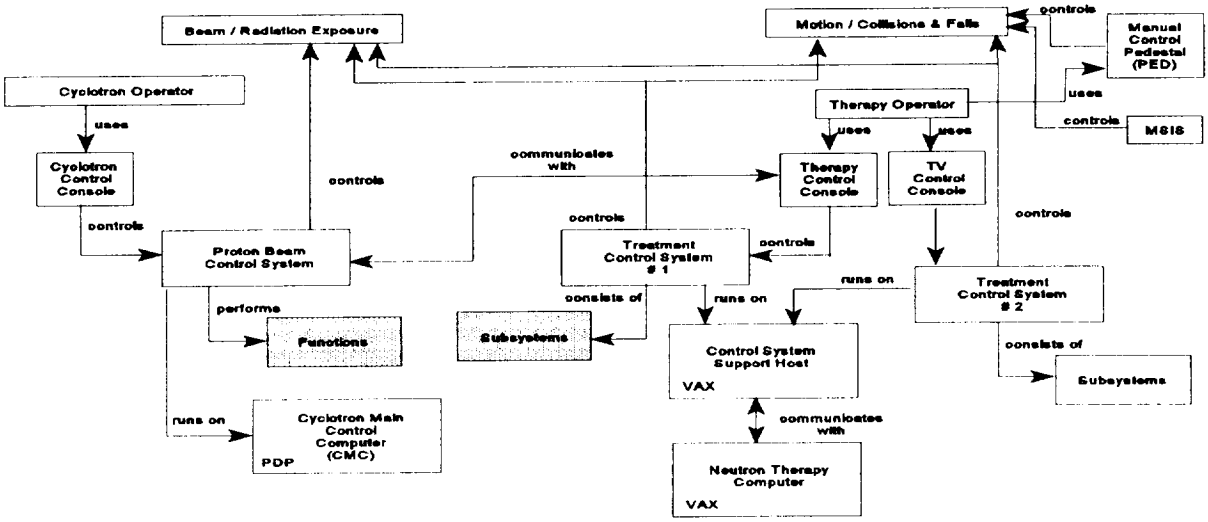

Figure 4.4. Control System Architecture

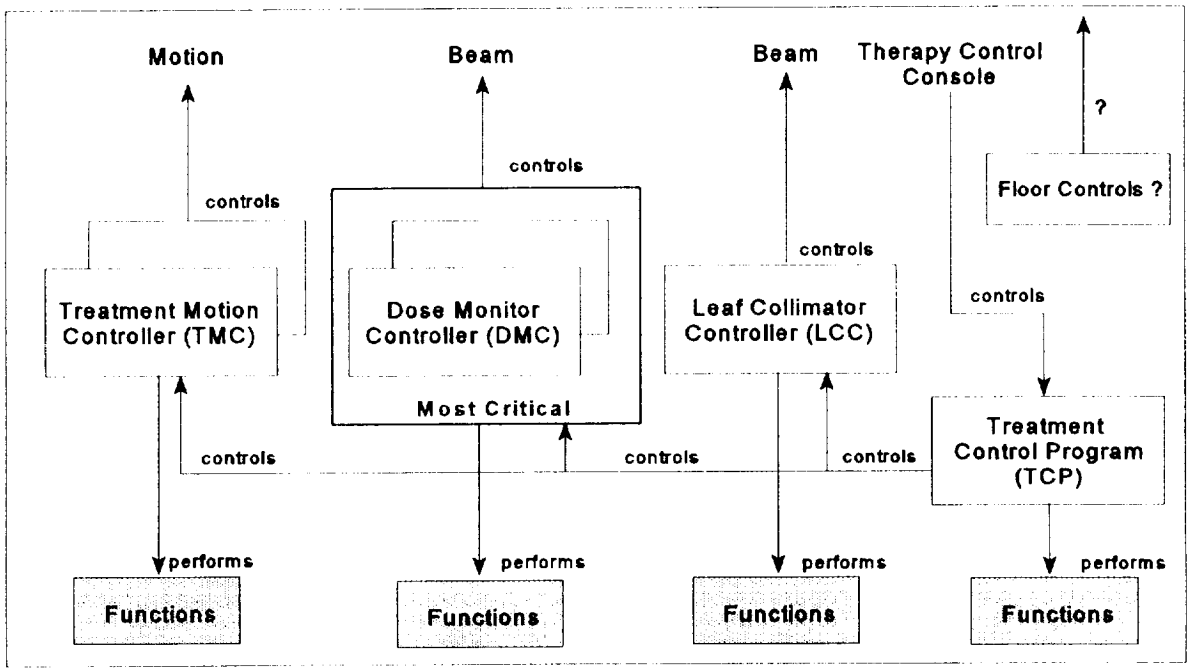

Figure 4.5. TCS Subsystem of the Control System

5 Future Directions

The use of formal methods for certification is a promising area; however, the level of confidence gained by having a formal specification of either a safety policy or of an individual safety-related software system is affected by issues such as

- Can expected complex, dynamic behavior be described by inputs, outputs, and states to an extent that actual behavior can be determined for both normal and faulted operational conditions?

- Can safety be expressed as a property or a set of properties, given current methods and semantics, that can be proven?

- Safety and the certification of safety requires a system solution. Can an approach and supporting semantics be developed that can integrate all the different system views and support the types of analysis required by each view? Can the need to make tradeoffs between all system requirements, such as safety, reliability, performance, security, etc., be supported?

- In the U. S. safety system, certification requirements are developed through a consensus process which determines that design and verification techniques specified in a standard are valuable and necessary. Can objective evidence be offered to demonstrate the added value of using formal methods?

As noted in Section 4.1 above, certification currently requires a large amount of documentation describing many various aspects of the design and implementation of a safety-related system. Figure 5.1 illustrates a view of how the many different ways of specifying various aspects of a system can be brought together into a safety certification process. As indicated in this figure, a primary driver of the process is a safety policy. A safety policy is a concept borrowed from the security domain and recommended by many in the safety-critical software community, including Leveson in [Leveson95].

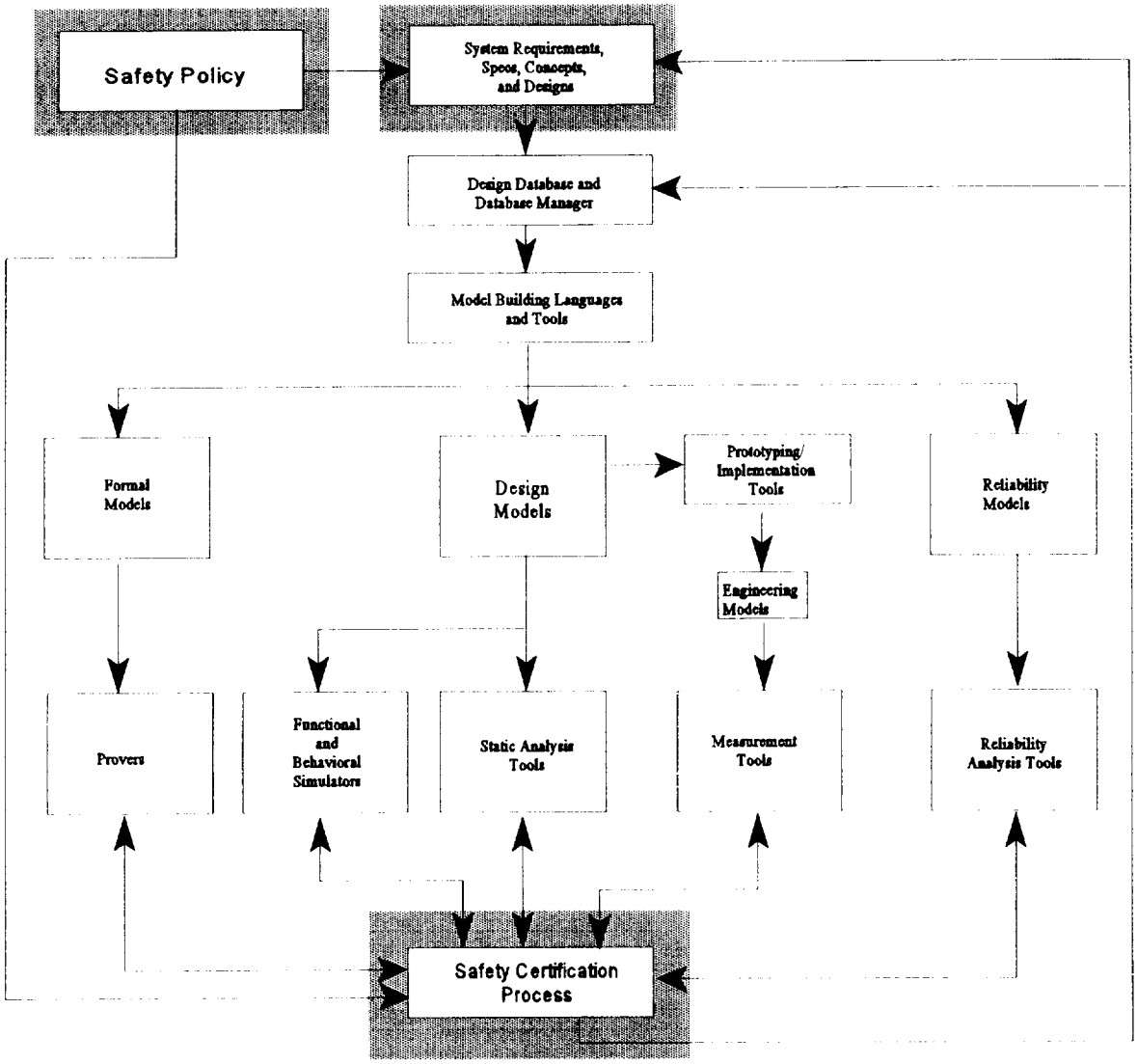

Figure 5.1. A View of a Programmable Systems Certification Process

Figure 5.2 illustrates the chain of development in the security domain from the setting of a safety policy to the implementation of code that is in compliance with that policy.

In the security domain, a security policy is devised that satisfies the requirements of applicable directives, installation settings, environmental assumptions, operations concepts and other

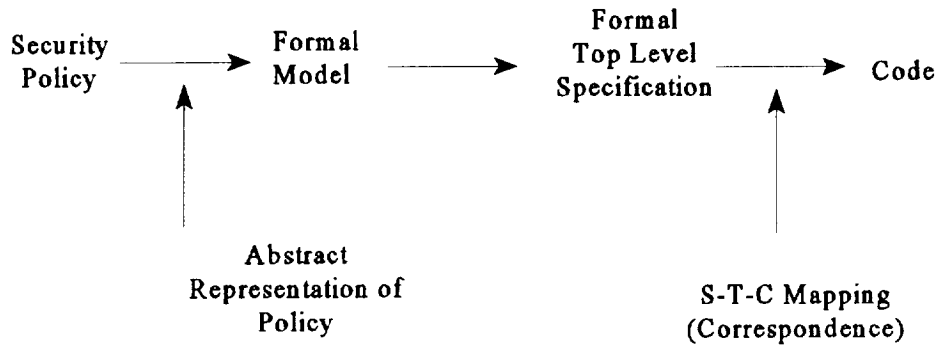

Figure 5.2. Formal Methods in the Security Domain

factors that constrain security considerations. An abstract representation of this policy is modeled in a formal model, which serves as the basis against which the subsequent development is evaluated for compliance with the security policy. The formal model and the formal top level specification are formally specified components and the formal top level specification can be proven to be consistent with the formal model. Thus, code correctly implemented from that specification will be guaranteed to be consistent with the security policy.

In the safety domain, however, there is not yet an accepted, similar construct to the security policy and its representation in a formal model. Thus, there is nothing against which a formal specification can be judged, and though code may be "proven" to be consistent with its specification, there is no way of being assured that the specification addresses all of the safety concerns. There are however, starts at defining security policies. An example of a general safety policy is given in [Leveson95]. As noted in Section 4.1 above for UL 1998, it is possible to view the requirements in a standard for safety-related systems as a safety policy.

The remaining components illustrated in Figure 5.1 would be developed and used to show compliance with the safety policy. It appears that in addition to the creation of formal models for certain aspects of the system specification and description, a necessary role for formal methods in this process would be to specify the relationships between all of the components and how they are used to certify the software. Integral to this process would be the use of a formal specification of each of the system components which would specify its essential characteristics (both in terms of evaluation criteria and representation styles), how it is modeled and

documented, and its role in the certification process. A further formal specification would link individual models and documents across development phases with the individual requirements of the certification process. This formal specification of the components of a certification process would provide several improvements to the way in which certification is currently performed. It would provide an increased level of consistency in the application of the process to all the various systems presented for certification; it would provide clear rationale to developers for all of the documentary evidence requested by the certifier; and it would provide a consistent and well-founded certification process.

References

Britt94 Joan J. Britt. "Case Study: Applying Formal Methods to the Traffic Alert and Collision Avoidance System (TACS) II". Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS '94), June 1994.

Heimdahl96 Mats P.E. Heimdahl and Nancy G. Leveson. "Completeness and Consistency in Hierarchical State-Based Requirements". *IEEE Transactions on Software Engineering*. Vol. 22, No. 6, pp. 363-377. June 1996.

Jacky90 Jonathan Jacky, Ruedi Risler, Ira Kalet, and Peter Wootton. *Clinical Neutron Therapy System: Control System Specification. Part I: System Overview and Hardware Organization*. Technical Report 90-12-01, University of Washington, Seattle, WA, December, 1990 (Revised March, 1991).

Jacky92 Jonathan Jacky, Ruedi Risler, Ira Kalet, and Peter Wootton, Jonathan Unger, Stan Brossard. *Clinical Neutron Therapy System: Control System Specification. Part II: User Operations*. Technical Report 92-05-01, University of Washington, Seattle, WA, May, 1992 (Revised December, 1992).

Jacky94a Jonathan Jacky and Jonathan Unger. *Designing Software for a Radiation Therapy Machine in a Formal Notation*. ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice, November 1994.

Jacky94b Jonathan Jacky and Jonathan Unger. *Formal Specification of Control Software for a Radiation Therapy Machine*. Draft Technical Report 94-07-01, University of Washington, Seattle, WA, July 1994.

Jacky95a Jonathan Jacky, Michael Patrick, and Ruedi Risler. *Clinical Neutron Therapy System: Control System Specification. Part III: Therapy Console Internals*. Draft Technical Report, University of Washington, Seattle, WA, August 16, 1995.

Jacky95b Jonathan Jacky and Jonathan Unger. *From Z to Code: A Graphical User Interface for a Radiation Therapy Machine*. September 1995.

Jacky95c Jonathan Jacky. "Specifying a Safety-Critical Control System in Z". IEEE Transactions on Software Engineering, Vol. 21, No. 2, pp.99-106. February 1995.

Jacky95d Jonathan Jacky, Michael Patrick, and Jonathan Unger. *Formal Specification of Control Software for a Radiation Therapy Machine (Revised)*. Technical Report

95-12-01, Radiation Oncology Department RC-08, University of Washington, Seattle, WA, December 1995.

Jacky96 Jonathan Jacky, Ruedi Risler, Jonathan Unger, Michael Patrick, and David Reid. *Experience Developing a Control Program for a Radiation Therapy Machine*. Technical Report 96-07-01, Radiation Oncology, University of Washington, Seattle, WA, August 1996.

Jaffe91 M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl, and B.E. Melhart. "Software Requirements Analysis for Real-Time Process-Control Systems". *IEEE Transactions on Software Engineering*, 17(3):241-258, 1991.

Knight93A John C. Knight. *Development of a Software Safety Process and A Case Study of Its Use*. Annual Report to NASA-LaRC for SEAS Proposal No. UVA/528344/CS93/103, University of Virginia, June 1993.

Knight93B John C. Knight and Darrell M. Kienzle. "Preliminary Experience Using Z to Specify a Safety-Critical System". *Proceedings of the Z User Workshop*. Edited by J. P. Bowen and J. E. Nicholls. Springer Verlag, 1993.

Leveson94 Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. "Requirements Specification for Process-Control Systems". IEEE Transactions on Software Engineering, Vol. 20, No. 9, September 1994 (pp. 684-707).

Modugno97 Francesmary Modugno, Nancy G. Leveson, Jon D. Reese, Kurt Partridge, and Sean D. Sandys. "Integrated Safety Analysis of Requirements Specifications". *Proceedings of the 3rd International Symposium on Requirements Engineering*. pp. 148 - 159. Annapolis, Maryland. January 1997.

UL1998 *UL 1998: Standard for Safety-Related Software*. First Edition. Underwriters Laboratories Inc. January. 1994.